

Projektowanie i implementacja

Wykład 7: Progresja gracza

mgr inż. Hubert Staniszewski

Czym jest progresja w grach?

- ▶ **Progresja** to proces, w którym gracz stopniowo zdobywa nowe możliwości, zasoby, poziomy, umiejętności lub dostęp do kolejnych treści.
- ▶ Może dotyczyć zarówno:
 - wzrostu **siły postaci**,
 - poszerzania **opcji decyzyjnych**,
 - jak i odkrywania **nowych obszarów, mechanik i wyzwań**.

Najprostsza definicja

Progresja to odpowiedź na pytanie:

„Co gracz ma więcej lub potrafi lepiej niż wcześniej?”

Systemy umiejętności

- ▶ **System umiejętności** określa, jakie zdolności gracz może odblokować, rozwijać lub modyfikować w trakcie gry.
- ▶ Jest to jeden z najważniejszych elementów progresji, ponieważ wpływa jednocześnie na:
 - styl gry,
 - tempo rozwoju postaci,
 - decyzje długoterminowe,
 - poczucie nagrody za postęp.
- ▶ Umiejętności mogą mieć charakter:
 - aktywny,
 - pasywny,
 - warunkowy,
 - wspierający inne mechaniki gry.

Rola systemu umiejętności

Dobry system umiejętności nie tylko wzmacnia postać, ale także **poszerza zbiór decyzji**, jakie gracz może podejmować w trakcie rozgrywki, walki lub eksploracji.

System Umiejętności



System umiejętności: System

- ▶ Typowy system umiejętności składa się z kilku warstw:
 - **definicji danych** umiejętności,
 - **warunków odblokowania**,
 - **logiki aktywacji**,
 - **kosztów i cooldownów**,
 - **efektów działania**,
 - **integracji z UI i zapisem gry**.

Pytanie implementacyjne

Jak zapisać umiejętność tak, aby była:

- ▶ rozszerzalna,
- ▶ testowalna,
- ▶ łatwa do balansu,
- ▶ niezależna od jednej konkretnej klasy postaci?

Jak zaprojektować umiejętność?

- ▶ Projektowanie umiejętności zaczyna się od pytania:
 - **jaką decyzję** ma dawać graczowi,
 - **jaki problem** ma pomagać rozwiązać,
 - **jaką fantazję działania** ma realizować.
- ▶ Dobra umiejętność powinna mieć jasno określone:
 - warunki użycia,
 - koszt lub ograniczenie,
 - efekt działania,
 - sytuacje, w których jest opłacalna,
 - relację do innych umiejętności i systemów gry.

Projektowe pytania kontrolne

- ▶ Kiedy gracz chce tego użyć?
- ▶ Co poświęca w zamian?
- ▶ Czy umiejętność daje nową opcję, czy tylko wzmacnia starą?
- ▶ Czy gracz rozumie jej wartość?

System Umiejętności



Błędy projektowe

- ▶ Najczęstszy błąd projektowy: tworzenie umiejętności, która jest tylko „większą liczbą”, a nie nową decyzją.
- ▶ Częstym błędem choć występującym rzadziej jest projektowanie umiejętności, które nie są przydatne poza chwilą otrzymania.

Jak napisać umiejętność?

- ▶ W kodzie umiejętność warto traktować jako:
 - **dane** opisujące jej parametry,
 - oraz **logikę wykonania**, która korzysta ze wspólnego systemu.
- ▶ Typowa implementacja obejmuje:
 - sprawdzenie warunków aktywacji,
 - pobranie kosztu,
 - uruchomienie efektu,
 - zapis cooldownu,
 - wysłanie informacji do UI i systemów pomocniczych.
- ▶ Dzięki temu wiele umiejętności może korzystać z jednej architektury wykonawczej.

Założenie

Jedna umiejętność nie powinna być osobnym, izolowanym systemem. Lepiej, gdy korzysta ze wspólnej architektury aktywacji i efektów.

Jak napisać umiejętność? - przykład aktywacji

```
1 if skill.CanActivate(user) then
2     skill.PayCost(user)
3     skill.Execute(user, target)
4     skill.StartCooldown()
5 end
```

- ▶ Najpierw sprawdzamy, czy umiejętność może zostać użyta.
- ▶ Następnie pobieramy koszt i wykonujemy efekt.
- ▶ Na końcu ustawiamy cooldown oraz aktualizujemy stan systemu.

Uwaga praktyczna

Nie warto pisać każdej umiejętności jako osobnego, zamkniętego skryptu. Lepiej mieć wspólny system aktywacji i zestaw efektów, które można łączyć konfiguracyjnie.

Progresja fabularna

- ▶ Nie każda progresja polega na wzroście siły postaci lub odblokowywaniu nowych umiejętności.
- ▶ W wielu grach równie ważna jest **progresja fabularna**, czyli stopniowe odkrywanie historii świata, relacji między postaciami czy nowych lokacji,
- ▶ Taki rodzaj progresji porządkuje doświadczenie gracza i nadaje sens kolejnym etapom rozgrywki.

Jak implementować progresję fabularną?

- ▶ Z punktu widzenia kodu progresja fabularna zwykle nie jest jednym systemem, lecz zbiorem **stanów i warunków**, które sterują dostępem do treści.
- ▶ Typowe elementy implementacji to:
 - **flagi fabularne** (questCompleted, metNPC, doorUnlocked),
 - **stany questów** i etapów misji,
 - **warunki dialogowe**,
 - **blokady dostępu** do lokacji, scen i wydarzeń,
 - **zapis decyzji gracza** wpływających na dalszy przebieg gry.
- ▶ Oznacza to, że progresja fabularna działa najczęściej jako system logiki warunkowej połączony z zawartością gry.

Przykład zapisu stanu fabularnego

```
1 struct NarrativeState {  
2     int mainQuestStage;  
3     bool metVillageElder;  
4     bool openedNorthGate;  
5     bool rescuedScholar;  
6     bool joinedRebels;  
7  
8     std::vector<std::string> completedQuests  
9     ;  
10    std::vector<std::string>  
11    unlockedLocations;  
12 };
```

- ▶ Taki model przechowuje najważniejsze informacje o stanie świata i postępie gracza.
- ▶ Nie każda zmienna musi być liczbą - często ważniejsze są:
 - flagi logiczne,
 - etapy questów,
 - lista odblokowanych elementów,
 - zapis podjętych decyzji.

Dobra praktyka

Stan fabularny powinien być zapisany jawnie. Nie warto ukrywać progresji w rozproszonych skryptach, bo utrudnia to debugowanie, rozwój gry i zapis stanu.

Przykład zapisu stanu fabularnego

```
1 enum class QuestState {
2     Locked,
3     Available,
4     InProgress,
5     Completed,
6     Failed
7 };
8
9 struct NarrativeState {
10     std::unordered_map<std::string, QuestState>
11         quests;
12     std::unordered_set<std::string> flags;
13     std::unordered_set<std::string>
14         unlockedLocations;
15     std::unordered_set<std::string> unlockedDialogs;
16 };
17
```

Cel tej struktury

Jedna struktura przechowuje stan wielu questów, zdarzeń i odblokowanych elementów świata.

Co przechowuje stan fabularny?

- ▶ Taki model nie opisuje jednego questu, lecz cały **stan progresji fabularnej**.
- ▶ Możemy w nim przechowywać:
 - statusy wielu questów,
 - ważne wydarzenia zapisane jako flagi,
 - odblokowane lokacje,
 - nowe opcje dialogowe i sceny.

Dobra praktyka

Progresja fabularna powinna być przechowywana w strukturze ogólnej i wielokrotnego użytku, a nie w pojedynczych, specjalnie pisanych zmiennych dla każdej misji.

Quest jako dane

- ▶ Definicja questa opisuje jego **strukturę**, a nie bieżący postęp gracza.
- ▶ Taki obiekt może zawierać:
 - identyfikator questa,
 - nazwę i opis,
 - warunki startu,
 - listę celów,
 - nagrody,
 - flagi ustawiane po ukończeniu.
- ▶ To są dane konfiguracyjne, więc dobrze nadają się do przechowywania jako:
 - Data Asset,
 - ScriptableObject,
 - Data Table,
 - plik JSON lub inny format danych.

Quest jako stan runtime

```
1 enum class QuestState {
2     Locked,
3     Available,
4     InProgress,
5     Completed,
6     Failed
7 };
8
9 struct QuestProgress {
10     std::string questId;
11     QuestState state;
12     int currentObjectiveIndex;
13     bool rewardGranted;
14 };
```

- ▶ Stan runtime opisuje aktualny postęp gracza w konkretnej sesji lub sejwie.
- ▶ To tutaj przechowujemy:
 - czy quest jest aktywny,
 - który etap trwa,
 - czy nagroda została przyznana,
 - co już zostało wykonane.
- ▶ Taki stan powinien być zapisywany w save data, a nie w definicji questa.

Różnica

Definicja questa jest stała, ale **QuestProgress** zmienia się wraz z działaniami gracza.

QuestManager

- ▶ Quest jako dane i stan runtime nie wystarczą, potrzebny jest system, który nimi zarządza.
- ▶ **QuestManager** odpowiada za:
 - uruchamianie questów,
 - sprawdzanie warunków startu i ukończenia,
 - aktualizowanie postępu,
 - ustawianie flag fabularnych,
 - przyznawanie nagród,
 - zapis i odczyt stanu.
- ▶ Dzięki temu logika questów nie jest rozrzucona po dialogach, NPC i triggerach świata.

Podział odpowiedzialności

- ▶ **QuestDefinition** - opis questa
- ▶ **QuestProgress** - aktualny stan
- ▶ **QuestManager** - logika obsługi

Progresja pionowa i pozioma

- ▶ Progresja gracza nie zawsze oznacza to samo.
- ▶ W projektowaniu gier często rozróżnia się dwa podstawowe kierunki rozwoju:
 - **progresję pionową**,
 - **progresję poziomą**.
- ▶ Oba podejścia wpływają na doświadczenie gracza, ale robią to w inny sposób.
- ▶ Jedno odpowiada głównie za **wzrost siły**, a drugie za **wzrost liczby dostępnych opcji i stylów gry**.

Progresja pionowa

- ▶ **Progresja pionowa** zwiększa skuteczność postaci w sposób bezpośredni.
- ▶ Gracz odczuwa ją jako realny wzrost siły i łatwiejsze pokonywanie wcześniejszych wyzwań.



Najczęściej oznacza:

- ▶ wyższe statystyki,
- ▶ większe obrażenia,
- ▶ więcej zdrowia,
- ▶ lepszy ekwipunek,
- ▶ dostęp do mocniejszych wersji znanych działań.

Progresja Pionowa



Progresja pozioma

- ▶ Zamiast zwiększać siłę gracza poszerza zestaw dostępnych możliwości:
 - nowych akcji,
 - alternatywnych narzędzi,
 - innych stylów gry,
 - dodatkowych sposobów rozwiązywania problemów.

Typowe efekty

- ▶ większa różnorodność decyzji,
- ▶ więcej stylów gry,
- ▶ większa elastyczność bez koniecznego wzrostu liczb.



Progresja pionowa: przykłady z gier

- ▶ **Diablo II / Diablo IV** - gracz zwiększa obrażenia, przeżywalność i skuteczność przez poziomy, ekwipunek oraz skalowanie buildów.
- ▶ **World of Warcraft** - rozwój postaci opiera się w dużej mierze na wzroście statystyk, poziomu oraz jakości wyposażenia.
- ▶ **The Witcher 3** - rozwój postaci obejmuje mocniejsze znaki, wyższe obrażenia i coraz skuteczniejsze narzędzia walki.
- ▶ **Skyrim** - mimo swobody rozwoju, znaczna część progresji polega na rosnącej skuteczności broni, magii i pancerza.

Progresja pozioma: przykłady z gier

- ▶ **Seria: Infamous** - pomimo minimalnego wzrostu siły w większej mierze gra nagradza gracza zestawem ruchów i możliwości podczas walki i eksploracji.
- ▶ **The Legend of Zelda: Breath of the Wild** - gracz zdobywa nowe sposoby interakcji ze światem, eksperymentuje z narzędziami i rozwiązuje problemy na różne sposoby.
- ▶ **Dishonored** - nowe moce nie zawsze oznaczają wyłącznie wzrost siły, ale przede wszystkim otwierają nowe style gry i sposoby przechodzenia poziomów.
- ▶ **Deus Ex** - rozwój postaci daje alternatywne metody działania: walkę, skradanie, hakowanie lub manipulowanie środowiskiem.

Typowe błędy projektowania progresji

- ▶ Progresja może łatwo przestać być satysfakcjonująca, jeśli system daje nagrody, które nie zmieniają realnie doświadczenia gracza.
- ▶ Najczęstsze błędy to:
 - zbyt wolny rozwój,
 - zbyt szybki rozwój,
 - iluzja wyboru,
 - nadmierny power creep,
 - nagrody bez znaczenia dla stylu gry,
 - przeciążenie gracza zbyt wieloma systemami naraz.
- ▶ Problem progresji bardzo często nie wynika z braku zawartości, lecz z braku **czytelnego sensu rozwoju**.

Pytanie kontrolne

Czy gracz po kolejnej godzinie gry naprawdę czuje, że jego postać lub możliwości uległy zmianie?

Błąd projektowy: iluzja wyboru

- ▶ **Iluzja wyboru** pojawia się wtedy, gdy gra oferuje graczowi wiele opcji, ale w praktyce tylko jedna lub dwie są naprawdę opłacalne.
- ▶ Problem może wynikać z:
 - złego balansu umiejętności,
 - zbyt słabych alternatyw,
 - dominującej strategii,
 - systemu, który premiuje tylko jeden styl gry.
- ▶ W efekcie gracz ma poczucie wyboru tylko pozornie, bo optymalna decyzja jest zawsze taka sama.

Przykład: drzewko umiejętności oferuje pięć opcji, ale jedna ścieżka daje wyraźnie najlepszy efekt w niemal każdej sytuacji.

Skutek

System wygląda na rozbudowany, ale nie daje realnej różnorodności decyzji.

Iluzja wyboru



Błąd projektowy: power creep

- ▶ **Power creep** to sytuacja, w której nowe elementy progresji są coraz silniejsze, przez co starsze opcje tracą znaczenie.
- ▶ Zjawisko to pojawia się często, gdy:
 - nowe umiejętności są projektowane jako „lepsze wersje” starych,
 - kolejne nagrody muszą stale robić większe wrażenie,
 - balans opiera się wyłącznie na wzroście liczb.
- ▶ W dłuższej perspektywie prowadzi to do destabilizacji systemu i wypierania wcześniejszej zawartości.

Skutek

Gracz przestaje korzystać z wcześniejszych narzędzi, bo nowe opcje są po prostu silniejsze, a nie ciekawsze.

Przykład: nowa umiejętność zadaje więcej obrażeń, kosztuje tyle samo i ma mniej ograniczeń niż poprzednia, więc starsza staje się zbędna.

Power Creep

